Git Like Version Control System

BY

Smitkumar Bhaveshkumar Patel (24C22053)
Shlok Jigarkumar Purani (24C22058)
Dhruv Kalpeshkumar Patel (24C22047)

Guided By
Prof. Madonna Lamin

A REPORT
SUBMITTED TO
ITM SLS Baroda University
in partial fulfillment of the requirements
for the subject
Data Structure and Algorithm

Computer Science & Engineering (with specialization in Artificial Intelligence & Data Science)

School of Computer Science Engineering & Technology

November 2025

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and appreciation to my supervisor, **Madonna Lamin** who has given me this bright opportunity to engage in the **Git like Version Control System**. It is my first step to establish a career in the **Data Structure & Algorithm**. I express my sincere gratitude to you.

I must say thanks to ITM SLS Baroda University's School of Computer Science Engineering & Technology all the staff for their continuous support and never ending encouragement throughout the course.

ABSTRACT

In the domain of modern software engineering, Configuration Management and Version Control are fundamental for ensuring data integrity across collaborative environments. This project presents the design and implementation of a **Distributed Version Control System (DVCS)**, architected to simulate the core operational mechanics of industry-standard tools such as Git. The system is built upon a robust **Client-Server architecture** utilizing **TCP/IP socket communication**, ensuring reliable, synchronous data transmission between remote clients and a centralized repository.

The primary technical objective of this research is to demonstrate the application of linear data structures in managing complex state transitions. Specifically, the system implements a **Last-In-First-Out (LIFO) Stack algorithm** to govern file history, enabling atomic **Undo/Redo operations** and precise state rollback capabilities. To support concurrent user sessions without latency or blocking, the server employs a **multi-threaded execution model**, assigning dedicated threads to manage independent client handshakes and command processing.

Functionally, the system provides a comprehensive suite of versioning utilities, including dynamic **Branching**, **Merging strategies**, and **Persistent Commits** to a physical disk storage. By integrating network protocols with efficient algorithmic state management, this project delivers a modular, scalable, and interactive solution that bridges the gap between theoretical Data Structures and practical, distributed system design.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

Abbreviation	**Full Form**
CLI	Command Line Interface
VCS	Version Control System

CHAPTER 1: INTRODUCTION

1.1 Problem Statement and Motivation

Problem Statement:

Modern software development relies heavily on version control systems to store, organize, and manage code changes efficiently. However, understanding the internal working principles of such systems—such as branching strategies, history stacks, conflict resolution, and network synchronization—can be challenging for beginners and system developers. To address this gap, there is a need for a simplified, modular, and customizable VCS model that replicates core functionalities without the overhead of full-scale implementations like Git.

Motivation:

Understanding version control architecture is fundamental for students and developers working in operating systems, data structures, and backend systems. However, the complexity of real-world VCS often makes it difficult to explore their internal mechanisms. By building a simplified Distributed VCS from scratch, learners can gain hands-on experience with socket programming, stack-based history management, threading, and command-line file operations. This project provides a practical learning platform to strengthen conceptual knowledge while developing programming and problem-solving skills.

1.2 Objectives

- To design a Distributed Version Control System using a client-server architecture.
- To implement core VCS operations such as **EDIT**, **UNDO** (revert changes), **REDO** (re-apply changes), and **COMMIT** (save to server).
- To enable branching functionality allowing users to create independent workspaces (BRANCH, CHECKOUT, MERGE).
- To implement a persistent storage mechanism where the "official" repository is saved to disk.
- To maintain metadata for each session, including active branches and history stacks.
- To ensure efficient handling of multiple users simultaneously using Multi-threading.
- To provide a user-friendly interface (CLI and GUI) to interact with the file system.

1.3 Project Scope and Direction

The project focuses on developing a simplified VCS, where each user session allows for independent file manipulation. The system will support essential commands including EDIT, UNDO, REDO, and COMMIT, enabling users to modify and save files. The scope includes implementing Stack data structures to manage the history of edits efficiently (LIFO). The project will also include a broadcast mechanism to notify all users when a commit occurs. The implementation will be kept modular, enabling future expansion or integration with more advanced functionalities.

CHAPTER 2: LITERATURE REVIEW

2.1 METHODOLOGY

The literature review for this project was conducted using a systematic and structured approach to identify, analyze, and synthesize relevant research and technical sources related to distributed systems, stack data structures, TCP/IP protocols, and CLI system design.

The following steps guided the review process:

1. Identification of Keywords

Relevant keywords and phrases were selected to locate scholarly articles and technical documents. Examples include:

- Distributed Version Control
- Stack Data Structures (LIFO)
- Socket Programming (TCP/IP)
- Client-Server Architecture
- Concurrency and Threading

2. Selection of Databases and Sources

The review relied on trusted academic and technical sources, including:

- Textbooks on Computer Networks and Data Structures
- Official Python Documentation (threading, socket, tkinter)
- Git Documentation (internal architecture)

2.2 SUMMARY

Author(s)/Year	Title/Source	Key Contribution	Relevance to Project
Torvalds, L. (2005)	Git Documentation	Explains distributed branching and merging.	Provides the logical foundation for BRANCH and MERGE implementation.
Python Software Fdn	Python Docs	Socket & Threading libraries.	Essential for implementing the client-server communication.

3. Inclusion and Exclusion Criteria

Inclusion Criteria

- Publications related to file system structure and design
- Articles discussing tree and graph data structures
- Research on operating system file management
- Technical reports on command-line utilities
- Materials published within the last 10–15 years (for modern relevance)

Exclusion Criteria

- Non-technical blogs without scientific basis
- Papers unrelated to file system architectures
- Outdated literature with obsolete technologies

4. Review and Analysis

Each selected paper or source was reviewed for:

- File system architectures and their evolution
- Data structures used in managing files and directories
- Algorithms for searching, creating, and deleting nodes
- CLI-based interaction models
- Memory management approaches in file systems

Comparative analysis was conducted to identify strengths, weaknesses, and design gaps in existing systems, helping justify the need for the proposed solution.

2.2 SUMMARY

Author(s)/Year	**Title/Source**	**Key Contribution**	**Relevance to current project**
anenbaum, A. (2015)	Modern Operating Systems	Explains internal structures of file systems, directory hierarchies, and file organization techniques.	Provides foundational theory for modeling a hierarchical tree-based file system.

CHAPTER 3 SYSTEM REQUIREMENTS

3.1 Introduction

The Distributed VCS project requires a combination of software and hardware resources to ensure smooth development, execution, and testing. Since the system simulates network operations and GUI rendering, it must be implemented in an environment capable of supporting multi-threading, socket binding, and graphical interfaces.

3.2 Software and Hardware Requirement

Minimum Hardware Requirements

- **Processor:** Dual-core CPU (2.0 GHz or above)
- **RAM:** 4 GB (to support multiple client threads)
- **Storage:** At least 200 MB free space for project files
- **Network:** Local Area Network (LAN) or Localhost capability.

Software Requirements

- **Development Environment:**
    - **Programming Language:** Python 3.x
- **Operating System:**
    - Windows 10/11, Linux, or macOS.
- **Supporting Tools:**
    - **IDE:** Visual Studio Code or PyCharm.
    - **Libraries:** socket, threading, tkinter.

3.3 Summary

This section outlines the necessary hardware and software to develop and run the VCS. These components create a reliable environment for successfully designing and implementing the system with essential operations such as EDIT, UNDO, REDO, and COMMIT.

CHAPTER 4: SYSTEM DESIGN

4.1 Introduction

System design defines the architecture, components, modules, interfaces, and data flow for the VCS. Since the project simulates version control operations such as branching, merging, and committing, a structured design is essential. The design uses a Centralized Server architecture where the server holds the master state and clients hold local workspaces.

4.2 Proposed System

The proposed system implements a multi-threaded server that handles concurrent user connections. It provides a command-line interface (CLI) that accepts user commands and executes the corresponding operations.

Key Features of the Proposed System

- **Branch Workspaces:** Each branch has its own history_stack and future_stack for independent Undo/Redo.
- **Core Operations:**
  - **EDIT –** Modifies current branch content.
  - **UNDO –** Reverts to previous stack state.
  - **COMMIT –** Saves branch to server file.
  - **SHOW –** Opens a GUI window to view server content.
- **Concurrency:** Uses Threading to allow multiple clients to connect without blocking.

4.3 Data Flow Diagram/Flowchart

1. Start & Initialization

- **Server Side:** The Server starts, binds to the configured Host/Port, and enters a listening state for incoming TCP connections.
- **Client Side:** The Client application launches and attempts to establish a socket connection to the Server.

2. Handshake & Session Creation

- The Client prompts the user for a Username.
- The Username is sent to the Server.
- The Server registers the user in the user_sessions map and assigns them to the default

'master' branch.

- The Server sends a "Welcome" message back to the Client.

3. Main Command Loop

- The Client displays a prompt (Command >) and waits for user input.
- The system parses the input to identify the Command Type.

4. Decision: Identify Command Type

- IF command = EDIT → Go to Edit Process.
- IF command = UNDO → Go to Undo Process.
- IF command = REDO → Go to Redo Process.
- IF command = COMMIT → Go to Commit Process.
- IF command = BRANCH → Go to Branch Creation Process.
- IF command = SHOW → Go to GUI Display Process.
- IF command = EXIT → Go to Termination.
- ELSE → Return "Unknown Command" error.

5. Edit Process (Data Input)

1. User inputs multi-line text (terminated by --END).
2. Client sends EDIT:<content> to Server.
3. Server accesses the user's Active Branch.
4. **Action:**

a) Push current content to History Stack.

b) Update branch content with new text.

c) Clear Future Stack (invalidates previous Redos).

5. Server returns "Update Successful" status.

6. Undo/Redo Process (Stack Manipulation)

- **Undo:**
    - Server checks if History Stack is empty.
    - If No: Pop state from History → Push to Future → Revert content.
- **Redo:**
    - Server checks if Future Stack is empty.
    - If No: Pop state from Future → Push to History → Apply content.
- Server returns the new current draft to Client.

7. Commit Process (Persistence & Broadcast)

- Server takes the content of the user's active branch.
- I/O Operation: Overwrites the server_repo.txt file on the hard drive.
- Broadcast: Server iterates through all connected sockets and sends a notification: *"[BROADCAST] User X performed COMMIT"*.
- Client receives confirmation.

8. GUI Display Process (SHOW)

- Client sends SHOW request.
- Server reads server_repo.txt and sends content with prefix SHOW_CONTENT:.
- Client detects prefix and launches a New Thread.
- Tkinter Window opens to display the file content (non-blocking).

9. Termination

- User enters EXIT.
- Client closes the socket connection.
- Server removes the user from connected_clients list and cleans up the thread.
- **End.**

4.4 Summary

Component	Description
Architecture	**Centralized Server:** Holds master state. **Clients:** Have local workspaces. **Multi-threaded:** Handles concurrent users.
Interface	**CLI:** Primary input for commands. **GUI (Tkinter):** Used strictly for the `SHOW` command to view files.
Key Features	**Branching:** Independent workspaces. **Stacks:** Separate `history_stack` (Undo) and `future_stack` (Redo) for every branch.
Workflow	**1. Connect:** Client connects via TCP. **2. Handshake:** User logs in and gets default branch. **3. Loop:** Cycle of Edit, Undo, Commit, or Show commands. **4. Exit:** Socket closes and threads cleanup.
Data Persistence	**Commit:** Saves active branch content to `server_repo.txt` and broadcasts the update to all connected users.

CHAPTER 5: IMPLEMENTATION

5.1 Introduction

The implementation phase converts the proposed system design into executable Python code. The system is implemented using Object-Oriented Programming (OOP) where the VCS manager, Branches, and Stacks are classes.

5.2 System Design

Stack Data Structure (for Undo/Redo): The history is managed using a LIFO (Last-In-First-Out) stack.

Class	Method	Description
Stack	push(item)	Adds content to the top.
	pop()	Removes top item (Undo).
	peek()	Views current state.

5.3 Algorithm/ Pseudocode

a) Undo Algorithm:

```
Function undo(username):
    branch = get_active_branch(username)
    IF history_stack is empty:
        Return "Nothing to undo"
    current_state = history_stack.pop()
    future_stack.push(current_state)
    branch.current_content = history_stack.peek()
    Return "Undo Successful"
```

b) Commit Algorithm:

```
Function commit(username):
    branch = get_active_branch(username)
    official_content = branch.current_content
    save_to_disk(official_content)
    broadcast_message("User committed changes")
```

5.4 Package/ Libraries

The implementation uses standard Python libraries:

Library	**Purpose**
socket	Network communication (TCP/IP).
threading	Handling multiple clients simultaneously.
tkinter	creating the GUI for the SHOW command.
os	File I/O operations for saving repositories.

CHAPTER 6: SYSTEM TESTING

6.1 Introduction

System testing is conducted to validate the functional correctness, stability, and performance of the VCS implementation. The goal is to ensure that all system features—such as branching, committing, and undoing—operate as intended under networked scenarios.

6.2 Test Cases

Test Case ID	Description	Input	Expected Output	Status
TC01	Connect to Server	Run client.py	"Welcome [User]!"	Pass
TC02	Edit File	EDIT: Hello World	"File updated..."	Pass
TC03	Undo Change	UNDO	Reverts to previous text	Pass
TC04	Create Branch	BRANCH: dev	"Branch 'dev' created"	Pass
TC05	Commit to Server	COMMIT	"Commit successful"	Pass

6.3 Test Results

The testing process confirmed the following observations:

- All file system operations performed as intended.
- The Stack correctly handled multiple Undo/Redo operations without underflow errors.
- The Broadcast system successfully notified other clients when a Commit occurred.
- The GUI (Tkinter) opened correctly on the SHOW command.

6.4 Performance Evaluation

- **Responsiveness:** Commands executed in real-time with negligible latency over Localhost.
- **Scalability:** The server successfully handled multiple concurrent client connections via threading.
- **Stability:** Stress testing with repeated EDIT and UNDO cycles showed no crashes.

6.5 Summary

The system testing phase validated the correctness, reliability, and performance of the VCS. All core functionalities were tested thoroughly. These results confirm that the implementation meets the project's goals.

CONCLUSION

The development of the **Distributed Version Control System** successfully demonstrates the fundamental concepts of network programming, data structures, and file management. By utilizing stacks for history management and sockets for communication, the project efficiently replicates the behavior of tools like Git. Key operations—including editing, undoing, branching, and committing—were implemented using systematic algorithms. The project contributes educational value by offering a practical implementation of client-server architecture and persistent data storage. Overall, the system meets all project objectives and demonstrates its effectiveness as a simplified simulation of a VCS suitable for academic learning.

REFERENCES

1. Python Software Foundation. *Python 3.10 Documentation: Socket Programming*.
2. Python Software Foundation. *Python 3.10 Documentation: Threading*.
3. Infosys
4. TryHackMe